



# Notion de réutilisabilité (1)

A ce point du cours,  
un programme n'est rien de plus qu'une  
séquence bien formée d'instructions simples ou  
composées (i.e. comprenant des structures de contrôle).

**Exemple:** soit  $P$  le programme d'affichage suivant<sup>1</sup>

```

cout << "*****" << endl;
cout << "Ceci est un petit texte" << endl
    << "    permettant    " << endl
    << "d'afficher quelque chose" << endl
    << "    à l'écran    " << endl;
cout << "*****" << endl;

```

Si la tâche réalisée par le programme  $P$  doit être exécutée plusieurs fois,  
une possibilité est de recopier  $P$  autant de fois que nécessaire...

... mais c'est [évidemment] une **très mauvaise** solution !



1. On pourrait également songer à non pas un programme d'affichage, mais à une fonction mathématique, telle que  $\text{SinC}(x)$ , soit  $f(x) = \frac{\sin(x)}{x}$



## Notion de réutilisabilité (2)

La duplication de larges portions identiques de code est à **poscrire**, car:

- ⇒ Cela rend la **mise à jour** du programme **fastidieuse**:  
il faut [manuellement] répercuter chaque modification de  $P$  dans chacune de ses copies
  - ☞ c'est par conséquent une **source supplémentaire d'erreurs**, et cela dégrade donc la fiabilité globale du programme.
  
- ⇒ Cela **réduit la lisibilité** globale du programme:  
la taille du programme source est plus importante, ce qui concourt à rendre les modifications et la traque des erreurs plus difficiles.
  
- ⇒ Cela **augmente** (de manière injustifiée<sup>2</sup>) la **taille** – donc l'occupation mémoire – du programme objet résultant.



Un langage de programmation doit donc fournir des moyens plus efficaces pour permettre la réutilisation de portions existantes de programmes.

2. Cette augmentation de taille est injustifiée **car** il existe une alternative à la duplication manuelle du code.



## Réutilisabilité: les fonctions (1)

Pour mettre en œuvre la **réutilisabilité**,  
la plupart des langages de programmation fournissent des  
mécanismes permettant de manipuler des **portions de programmes**.

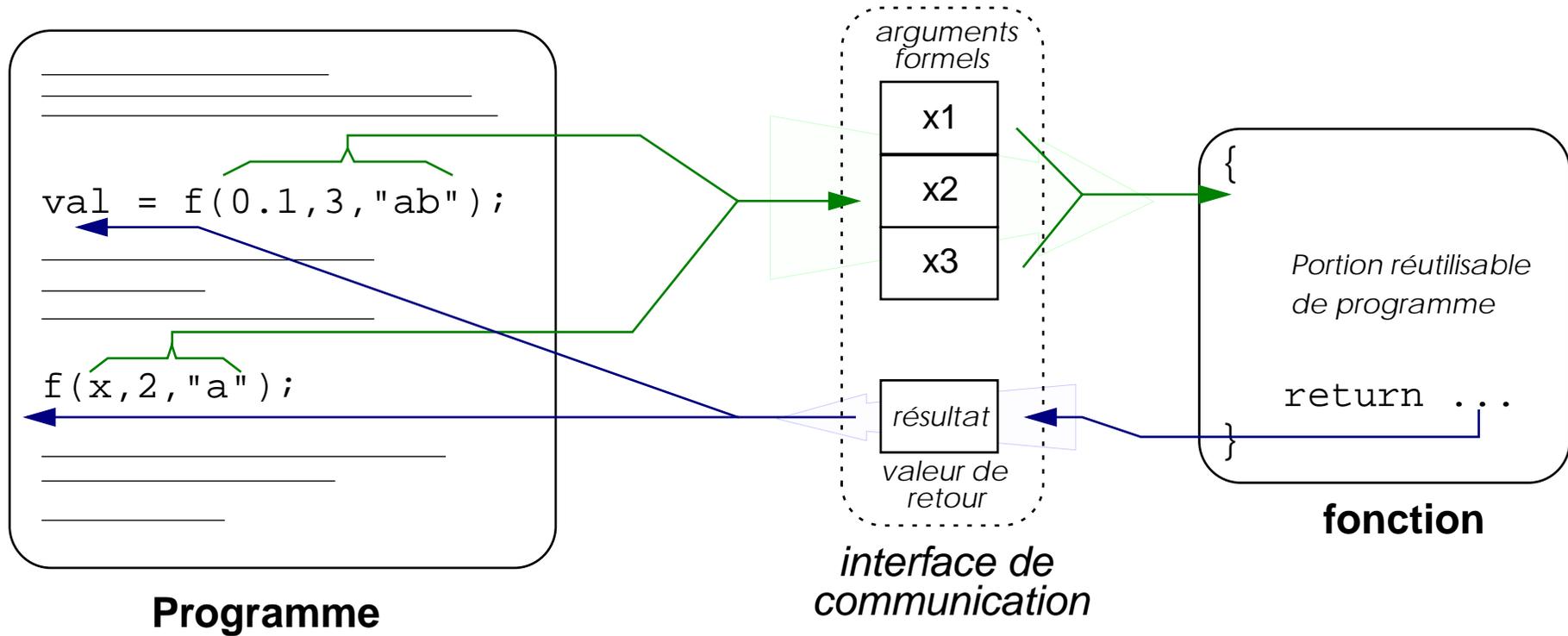
En C++, ces mécanismes vont s'appuyer sur la notion de *fonction*

De façon imagée, une fonction est assimilable à une expression,  
dont l'évaluation (on dit l'**appel**) correspond à l'exécution de la portion  
de programme qui lui est associée, et produit une valeur (résultat de l'évaluation),  
la *valeur de retour* de la fonction.



# Réutilisabilité: les fonctions (2)

Une **fonction** est donc une **portion réutilisable** de programme, associée à la définition d'une **interface explicite** avec le reste du programme (permettant de définir la manière d'utiliser cette fonction), par le biais d'une **spécification des arguments formels** (les «entrées») et de la **valeur de retour** (les «sorties»).





## Réutilisabilité: les fonctions (3)

Plus précisément, une fonction est un élément informatique caractérisé par:

- un **identificateur**, qui n'est autre qu'une référence particulière à l'élément «fonction» lui-même;
- un **ensemble d'arguments formels**, qui correspondent en fait à un ensemble de références formelles à des entités définies à l'extérieur de la fonction, mais dont les valeurs seront potentiellement utilisées par la fonction;
- un **type de retour**, qui permet de préciser la nature du résultat retourné par la fonction
- un **corps**, qui correspond à un bloc (la portion de code réutilisable qui justifie la création de la fonction) et qui induit également une portée –locale– pour les éléments (variables, constantes) définis au sein de la fonction.



## Prototypage (1)

Tout comme dans le cas des variables, la mise en œuvre d'une fonction se fait en 2 étapes:

- le *prototypage* ou *spécification* (équivalent de la déclaration des variables)
- la *définition* ou *implémentation* (à peu près équivalent de l'initialisation)

La syntaxe d'un *prototypage* est:

```
<type> <identificateur>(<arguments>);
```

Où *type* est le type de la valeur de retour renvoyée par la fonction<sup>3</sup>,  
*identificateur* est le nom donné à la fonction, et  
*arguments* est la liste des arguments formels, de la forme:

```
type1 id-arg1, type2 id-arg2, ..., typen id-argn
```

**Exemple:** `float puissance(const float base, float exposant);`

3. On dit souvent (par extension), le «type de la fonction»

## Prototypage (2)



Dans les prototypes de fonctions, les identificateurs des arguments sont optionnels, mais servent généralement à rendre plus lisible le prototype.

Ainsi, la fonction `puissance` spécifiée précédemment pourrait être prototypée (de manière équivalente pour le compilateur, mais nettement moins lisible) par:

```
float puissance(const float, float);
```

Comme dans le cas des variables, les fonctions ont une portée (et une visibilité). Ceci implique que toute fonction doit être prototypée avant d'être utilisée.

Les prototypes des fonctions seront ainsi toujours placés avant le corps de la fonction *main*.

### Exemple:

---

```
float puissance(const float base, float exposant);
void main()
{
    ...
    x = puissance(8.0, 3.);
    ...
}
```

---



## Définition (1)

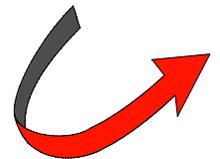
La *définition* permet «l'affectation» d'un corps (bloc d'instructions) au prototype de la fonction.

La syntaxe d'une **définition** est:

```
<type> <identificateur>(<arguments>)  
{  
    <corps de la fonction>  
    [return <valeur>]  
}
```

Où *corps de la fonction* est le **bloc** définissant la fonction, et dans lequel les arguments formels jouent le rôle de **variables ou constantes locales**, initialisées avec la valeurs des arguments employés lors de *l'appel* de la fonction.

Une instruction de rupture de séquence particulière, **return**, permet de terminer la fonction en retournant comme résultat de l'exécution la valeur spécifiée par *valeur*.





## Définition (2)

### Exemple:

```

#include <iostream>

// Prototypages .....

float moyenne(const float x1, const float x2);

// Définitions .....

void main()
{
    cout << moyenne(8.0,3.0) << endl;
}

float moyenne(const float x1, const float x2)
{
    return (x1+x2)/2.0;
}

```



## Définition (3)

Comme dans le cas des variables, il est possible de réaliser le prototypage et la définition **en une seule étape** (dans ce cas la syntaxe est simplement celle de la définition)

On peut donc directement écrire:

```
float moyenne(const float x1, const float x2)
{
    return (x1+x2)/2.0;
}
void main()
{
    cout << moyenne(8.0,3.0);
}
```

Cependant, si pour les autres types de données il est recommandé de toujours initialiser lors de la déclaration, il est préférable, pour les fonctions, de réaliser le prototypage et la définition en 2 opérations distinctes.<sup>4</sup>

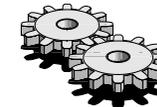
4. L'élément prédominant est de rendre le code le plus lisible possible. A cet fin, il est souvent préférable de regrouper les différentes déclarations en un endroit unique (typiquement en début de programme), et disposer ainsi d'une sorte d'index dans lequel il est aisé de retrouver le prototype d'une fonction spécifique.



## Evaluation d'un appel fonctionnel (1)

Pour une fonction  $f$  définie par: `int f(t1 x1, t2 x2, ..., tn xn) { ... }`  
 l'évaluation de *l'appel fonctionnel* `f(arg1, arg2, ..., argn)`  
 s'effectue de la façon suivante:

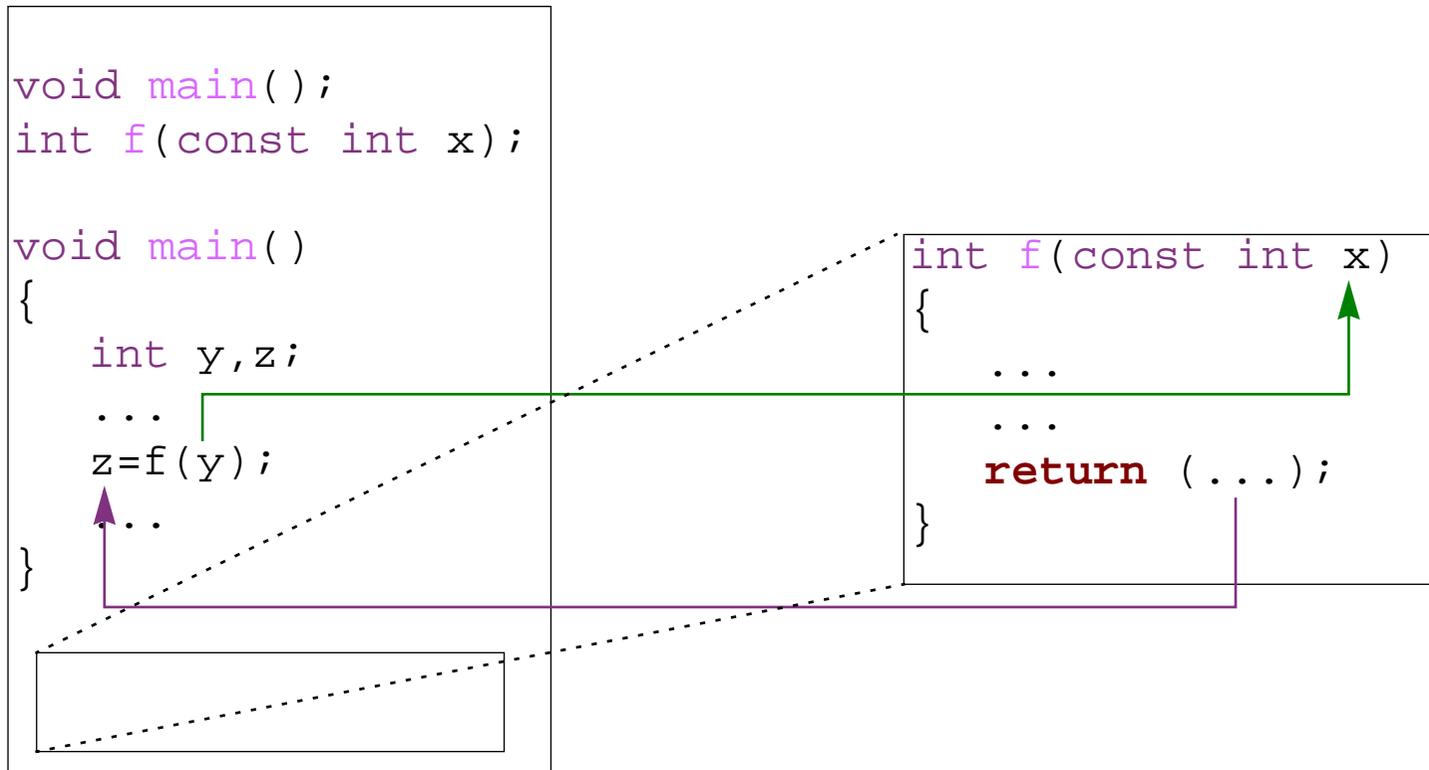
- (1) les arguments  $arg_1, arg_2, \dots, arg_n$  sont évalués (de la gauche vers la droite),  
 et les valeurs produites sont liées avec les arguments formels  $x_1, x_2, \dots, x_n$  de la fonction  $f$   
 (cela revient concrètement à réaliser les affectations:  $x_1=val_1, x_2=val_2, \dots, x_n=val_n$ ),  
 où  $val_1, val_2, \dots, val_n$  sont le résultat de l'évaluation des arguments ( $val_i \leftarrow (arg_i)$ );
- (2) le bloc correspondant au corps de la fonction  $f$  est exécuté;
- (3) l'expression (entière dans notre cas) définie par le mot clef `return` est évaluée,  
 et est retournée comme résultat de l'évaluation de l'appel fonctionnel.





## Evaluation d'un appel fonctionnel (2)

L'évaluation de l'appel fonctionnel peut être schématisé de la façon suivante:





## Evaluation d'un appel fonctionnel (3)

Pour la fonction `moyenne()` précédemment définie, soit l'appel fonctionnel:

```
moyenne((sqrt(2*3*4)+1), 12%5)
```

- 1 Evaluation des arguments et liaison avec les arguments formels:

```
x1 = (sqrt(2*3*4)+1) => x1 = 5.89897948557
```

```
x2 = 12%5 => x2 = 2.0
```

- 2 Exécution du corps de la fonction:

```
moyenne = (x1+x2)/2.0 => moyenne = 3.94948974278
```

- 3 La valeur de retour, correspondant à l'évaluation de l'expression «appel fonctionnel», est donc: 3.94948974278

L'appel est alors équivalent à l'exécution du bloc:

```
float moyenne;  
{  
    const float x1(sqrt(2*3*4)+1);  
    const float x2(12%5);  
    moyenne = (x1+x2)/2.0;  
}
```

# Fonctions sans valeur de retour



Il est également possible de définir des fonctions **sans valeur de retour** (i.e. des fonctions qui ne renvoient rien, et donc à peu près équivalentes à de simple sous-programme).

Cette absence de valeur de retour sera indiquée, dans le prototype et la définition de la fonction, par un type de retour particulier, le type **void**

Dans ce cas, le mot réservé `return` n'est pas requis, l'exécution de la fonction se terminant à la fin du bloc constituant son corps. Il est toutefois possible l'instruction `return` (dans ce cas sans indication d'expression de retour), afin de provoquer explicitement (et généralement prématurément) la terminaison de la fonction.

## Exemple:

```
void afficheEntiers(const int n)
{
    for (int i(0); i<n; ++i)
        cout << i << endl;
    return;
}
```

```
void main()
{
    int n(10);
    afficheEntiers(n+1);
}
```



## Fonctions sans arguments

De même, il est possible de définir des fonctions **sans arguments**.

Dans ce cas, la liste des arguments du prototype et de la définition est tout simplement vide (mais les parenthèses délimitant cette liste vide doivent être présentes)

### Exemple:

```
int saisieEntiers();

void main()
{
    int val = saisieEntiers();
    cout << val << endl;
}

int saisieEntiers()
{
    int i;
    cout << "Entrez un entier: ";
    cin >> i;
    return i;
}
```



## Structures de données: types composés



Les **types élémentaires** (nombres entiers, réels, caractères, booléens, ...) permettent de représenter des **concepts simples** du monde réel: des dimensions, des sommes, des tailles, ...

Les identificateurs associés permettent de donner une signification aux divers entités manipulées (pour cette raison, il faut toujours utiliser des noms aussi explicites que possible)

### Exemples:

```
age = 18; poids = 62.5; taille = 1.78; ...
```

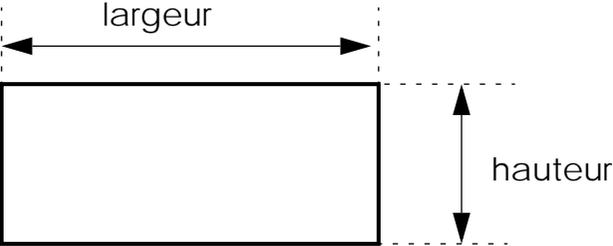
Cependant, de nombreux concepts plus sophistiqués ne se réduisent pas à une seule entité élémentaire...





## Exemple de type composé

Un rectangle par exemple peut être définis par 2 grandeurs élémentaires: sa «largeur» et sa «hauteur».



Si l'on ne dispose que de type élémentaires, ces grandeurs seront représentée indépendamment l'une de l'autre, et leur relation devra être indiquée par le programmeur, par exemple au moyen d'identificateurs tels que: `rectangle_largeur` et `rectangle_hauteur`.

### Rectangle

<i>longueur</i>	<i>hauteur</i>
20	1.75

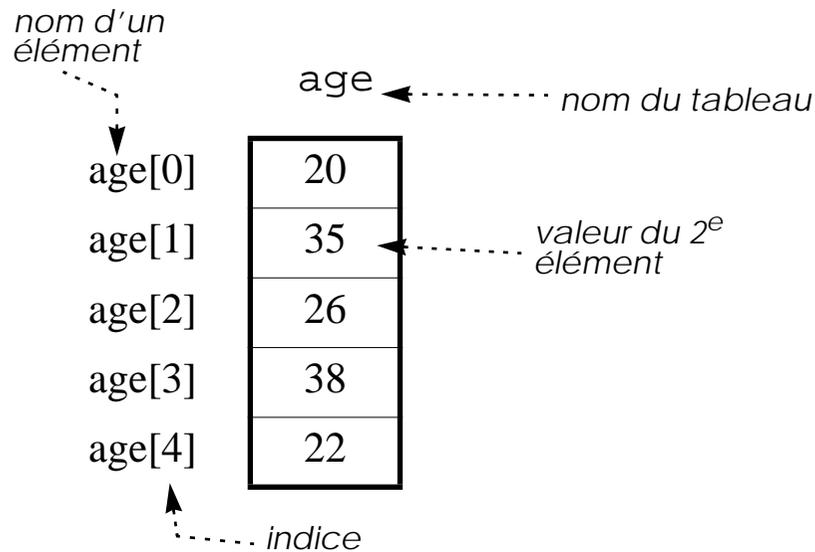
Afin d'éviter au programmeur d'avoir à assurer la cohésion des entités composites, un langage de programmation doit fournir le moyen de combiner les type élémentaires pour construire et manipuler des objets informatiques plus complexes.

Ces types non élémentaires sont généralement appelés **types composés**.



# Types composés: les tableaux

Une première façon de composer des types élémentaires est de regrouper une collection d'entités (de même type) dans une structure tabulaire: le *tableau*



En C++, les tableaux sont des ensembles d'un **nombre fixe** d'éléments:

- de même type, appelé *type de base* (le tableau est donc une structure *homogène*);
- référencés par un même identificateur, **l'identificateur du tableau**;
- individuellement accessibles par le biais d'un **indice**.



Le type de base est quelconque: il peut s'agir de n'importe quel type, élémentaire ou composé. On pourra donc définir des tableaux d'entiers, de réels, de caractères, ... et de tableaux.



Un tableau de taille fixe peut être déclaré selon la syntaxe suivante:

```
<type> <identificateur>[<taille>];
```

Avec *type* n'importe quel type, élémentaire ou non,  
et *taille* une valeur entière littérale ou une expression entière constante  
(i.e. évaluable lors de la compilation)

Cette instruction déclare un tableau désigné par *identificateur*,  
comprenant *taille* éléments de type *type*.

## Exemple:

Ainsi, le tableau *age* vu précédemment  
peut être déclaré par:

```
int age[5];
```

	age
age[0]	20
age[1]	35
age[2]	26
age[3]	38
age[4]	22



## Tableaux: *déclaration-initialisation*

Naturellement, un tableau de taille fixe peut être **initialisé** directement, lors de sa déclaration.

La syntaxe est alors:<sup>5</sup>

$$\langle type \rangle \langle identificateur \rangle [\langle taille \rangle] = \{ \langle val_1 \rangle, \dots, \langle val_n \rangle \};$$

**Exemple:**

```
int age[5] = {20, 35, 26, 38, 22};
```

Remarquons que dans le cas d'une *déclaration-initialisation*, la spécification de la taille du tableau est **optionnelle**.

Ainsi, la déclaration-initialisation précédente peut également être écrite:

```
int age[] = {20, 35, 26, 38, 22};
```

5. Il est également possible de n'initialiser que les premiers éléments du tableau, comme dans: `<int age[5] = {20, 35}>`.



## Tableaux: accès aux éléments

Chaque élément du tableau est **accessible individuellement**.

Pour cela, on utilise un *indice*<sup>6</sup> placé entre crochets « [ ] », indiquant le rang de l'élément dans le tableau<sup>7</sup>.



Les éléments d'un tableau de taille  $n$  sont numérotés de 0 à  $n-1$ .

**Il n'y a pas de contrôle de débordement du tableau !**

### Exemple:

L'accès au premier élément du tableau `age` s'obtient au moyen de l'expression: «`age[0]`», et l'accès au dernier élément par «`age[4]`», comme dans:

```
age[0] = 3;  
age[4] = age[1+2]; // soit age[4] = age[3]
```

6. Indice qui peut être une expression numérique.

7. Pour être tout à fait précis, on n'accède pas aux éléments en spécifiant un rang, mais un déplacement (une distance, dont l'unité est l'élément/expri-  
mée en nombre d'éléments), relativement au premier élément du tableau. C'est pour cela que l'indilage débute à 0 et non à 1.



## Tableaux: tableaux de caractères (1)

Les tableaux de caractères forment une catégorie particulière de tableaux. Ils sont en particulier utilisés pour représenter les **chaînes de caractères** (délimitées par les guillemets anglais " ").

Ainsi, une définition tel que:

```
char mot[7] = {'b', 'o', 'n', 'j', 'o', 'u', 'r'};
```

*pourrait* être utilisée pour représenter la chaîne "bonjour".

Cependant, comme une chaîne de caractères étant un élément de taille variable, il faut lui associer une information permettant d'en retrouver la longueur.

A l'inverse de langages comme *Pascal*, C++ ne code pas explicitement cette longueur, mais délimite la chaîne par un caractère spécial – « '\0' » –, marquant la fin de la chaîne.

La chaîne "bonjour" sera donc implémentée par:

```
char mot[8] = {'b', 'o', 'n', 'j', 'o', 'u', 'r', '\0'};
```

et sa taille sera de **8 caractères**, et non pas 7 !



## Tableaux: tableaux de caractères (2)

La syntaxe de C++ autorise cependant une initialisation **plus pratique** des «tableaux-chaîne de caractères»:

```
char mot[] = "bonjour";8
```

Une telle représentation des chaînes, héritée du langage C, à comme inconvénient de **fixer la taille de la chaîne** de caractères.



Cette manière de procéder impose un certain nombre de contraintes lors d'opérations sur les chaînes de caractères (par exemple l'affectation, la concaténation de deux chaînes, etc.)

Ceci non seulement diminue l'efficacité de ces traitements, mais surtout oblige les programmeurs à effectuer eux-mêmes un certain nombre de manipulations dans le cas d'opérations sur les chaînes qui ne sont pas fournies dans les bibliothèques standards.

Pour pallier ce défaut, on pourra utiliser, comme nous le verrons plus loin dans le cours, un type issu de l'approche objet de C++, le type **string**

8. Cette instruction est effectivement équivalente à celle vue précédemment (incluant le caractère '\0')



## Tableaux: tableaux multidimensionnels (1)

Le type de base d'un tableau est peut être un type quelconque, y compris composé. En particulier, le type de base d'un tableau peut être lui-même un tableau.

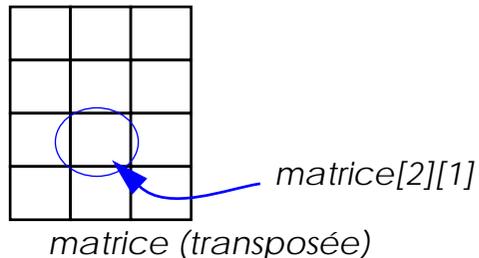
La syntaxe est alors:

```
<type> <identificateur> [<dim1>] [<dim2>] [...] [<dimn>];
```

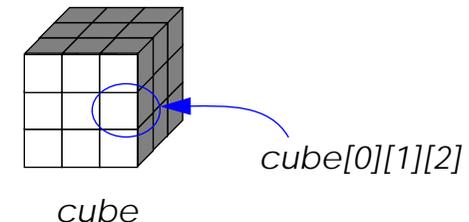
Les entités d'un tel type sont alors des tableaux de tableaux, soit des tableaux multidimensionnels<sup>9</sup>.

### Exemple:

```
int matrice[4][3];
```



```
float cube[3][3][3];
```



9. En fait, les tableaux informatiques correspondent aux vecteurs mathématiques, et les tableaux de tableaux à des matrices.

## Tableaux: tableaux multidimensionnels (2)



Les tableaux multidimensionnels sont naturellement initialisables lors de leur déclaration, et chaque éléments est accessible individuellement.

### Exemple:

```
int matrice[4][3] = {{0,1,2},{3,4,5},{6,7,8},{9,10,11}};
matrice[2][1] = 70;
```

	<i>matrice[][j]</i> →		
<i>matrice[i][]</i> ↓	0	1	2
	3	4	5
	6	70	8
	9	10	11

Dans les deux cas, il faut spécifier autant d'indices qu'il y a de dimensions dans le tableau.



## Tableaux: typedef (1)

L'utilisation de variables dans un programme implique généralement que l'on spécifie plusieurs fois le type de ces variables (lors de la déclaration des variables, lors du prototypage des fonctions qui en font usage, lors de conversions, etc...)

Lorsque le type est complexe, sa définition peut être ardue, et est généralement longue, ce qui ne facilite pas la lecture du programme. En outre, les modifications éventuelles à apporter à la définition du type doivent être opérées sur chacune de ses occurrences. Comme dans le cas des «blocs d'instructions réutilisables» (i.e. les fonctions), et pour les mêmes raisons, la duplication de la définition d'un type est à éviter.

La commande `typedef` permet pour cela de définir des *synonyme (alias)* de types, qu'ils soient fondamentaux ou composés:

```
typedef <type> <alias>;
```

Cette instruction permettra de désigner le type `type` indifféremment par `type`, ou au moyen de l'identificateur `alias` (`typedef` n'introduit pas de nouveau type, mais un nouveau nom pour le type)



## Tableaux: typedef (2)

### Exemple

```
typedef int longueur;  
longueur diametre;  
int rayon;
```

Dans cet exemple, diametre et rayon sont de même type (int). Mais, si l'on utilise longueur pour exprimer toutes les longueurs, et que pour une raison quelconque on soit amené à changer la représentations des longueurs (par des entiers positifs, ou par des réels), il suffira d'opérer ce changement dans la définition de l'alias longueur, plutôt qu'à chaque occurrence de int censée représenter une longueur.

Dans le cas des tableaux, et notamment les tableaux multidimensionnels, on peut se servir avantageusement de cette commande pour rendre plus explicite les déclarations et les usages ultérieurs:

```
typedef int vecteur2[2];           // définit le type vecteur2  
                                  // comme un tableau de 2 entiers  
typedef vecteur2 matrice3x2[3];  // définit le type matrice3x2  
                                  // comme un tableau de 3 vecteur2  
matrice3x2 A = {{1,2},{3,4},{5,6}}
```